lecture 11

MIPS assembly language 4

- functions
- MIPS stack

February 15, 2016
## MIPS registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Number</th>
<th>Usage</th>
<th>Preserved on call</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
<td>n.a.</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for the assembler</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>value for results and expressions</td>
<td>no</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments (procedures/functions)</td>
<td>yes</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved</td>
<td>yes</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$k0-$k1</td>
<td>26-27</td>
<td>reserved for the operating system</td>
<td>n.a.</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
<td>yes</td>
</tr>
</tbody>
</table>
functions in C

```c
main() {
  int m, n
  m = myfunction(n);
}

int myfunction(int j) {
  int k;
  return k;
}
```
parent must:

- save in Memory any registers it will need later (after return)
- provide arguments to child
- provide return address to child (so child can jump back when done)
- branch to child
child must:

- allocate space for local variables (registers, memory) and not write over parent's data (registers, memory)
- compute value and return it to parent
- branch back to parent
parent must:
  
  - branch to child

child must:
  
  - branch back to parent

How?
Jump and Link: Jump Register

main:

jal myfunction

# writes address of next instruction into
$ra = $31

myfunction:

jr $ra
J format

offset address of instruction to jump to
(defined a few lectures from now)

R format
Provide argument(s), return value(s)

How?

$\text{argument registers}$

$ao, al, a2, a3$

$\text{return value registers}$

$v0, v1$

$2, 3$
```c
main() {
    int m, n;
    m = myfunction(n);
}

int myfunction(int j) {
    int k;
    return k;
}

provide argument(s)
move $a0, $s0
jal myfunction
move $s3, $v0

return value(s)
move $v0, $s1
jr $ra
```
Encapsulation Problem

- The author of the parent function might not know which `$s` and `$t` registers the child uses.
- The author of the child function might not know which `$s` and `$t` registers the parent uses.
Kitchen Policies

1. By default, all dishes are clean. Use dishes, then clean them.

2. By default, all dishes are dirty. Wash what you need and leave them dirty.

Problems arise when two housemates have different policies.
MIPS register conventions (policies)

- parent (caller):
  - assume that $s0, ... $s7 will contain same values before and after call
  - don’t assume that $t0, ... $t7 will contain same values after call

(If parent will need values in $t0, ... $t7 after call, then these values must be stored in Memory prior to call, and loaded after call.)
MIPS register conventions

\textbf{child (callee)}

\begin{itemize}
  \item assume that $t0, \ldots, t7$ are not being used by parent.
  \item assume that $s0, \ldots, s7$ are being used by parent.
\end{itemize}

(If child needs to use any of $s0, \ldots, s7$, then it first needs to store the current values in Memory, and re-load the values prior to returning to parent.)
How and where do functions store register values in memory?
0x80000000

Stack

"heap"

user instructions

user data

0x10000000

0x10000000

function arguments, local variables, return address, etc

see lecture 10 (also "malloc" for those who know C)
Stack frames

0x80000000

stack

main

0x10000000

heap

my function

0x100000000
main stores temporary registers before calling myfunction. (Here we assume variables m and n use $t0 and $t1.)

myfunction stores previous values of any save registers that it will use. (Here we assume variable k uses $s0.)
Stack pointer (register $sp = $29) contains the lowest address of the stack.
parent:

```
addi $sp, $sp, -8
sw $t1, 0($sp)
sw $t0, 4($sp)
jal child
lw $t1, 0($sp)
lw $t0, 4($sp)
addi $sp, $sp, 8
```
allowed

\[
\begin{align*}
\text{parent}: & \\
\text{sw} & \Rightarrow t0, -4(sp) \\
\text{sw} & \Rightarrow t1, -8(sp) \\
\text{addi} & \Rightarrow sp, sp, -8 \\
\text{jal} & \Rightarrow \text{child} \\
\text{addi} & \Rightarrow sp, 8(sp) \\
\text{lw} & \Rightarrow t1, -8(sp) \\
\text{lw} & \Rightarrow t0, -4(sp) \\
\end{align*}
\]
Child:

```
addi $sp, $sp, -4
lw $s0, 0($sp)
addi $sp, $sp, 4
jr $ra
```

do work
here using
register $s0
When functions call another, we need to ensure that data is safe (not written over).
When a function is both a child and a parent, it must store the return address (to its own parent) before it calls its child. It also needs to store $a$ registers (from parent) that it modifies.

(See conventions on slide 2).
(Static, compile time)

\[ f_1 \]
\[ f_2 \quad \leftrightarrow \quad f_3 \]

(recursive)

(dynamic, runtime)

\[ f_1 \]
\[ f_2 \]
\[ f_2 \quad f_3 \]

"Call tree"

("pre-order traversal" - COMP 250)
Call tree

Evolution of stack over time
$s0, \ldots$

$to, \ldots$

$ao, \ldots$

$ra$

stack

frame in
general case
Example: recursion

```java
int sumToN (int n) { // n > 0
    if (n == 0)
        return 0;
    else
        return n + sumToN(n-1);
}
```
sumton:
# if n == 0 then branch to base case
# else push the two items onto the stack:
#   - return address ($ra)
#   - argument n ($a0)
#
# compute (n-1) argument for next call
# jump and link to sumton (recursive)

# load the return address (pop)
# load argument from the stack (pop)
# change the stack pointer

# register $v0 contains result of sumton(n-1)
# add argument n to $v0
# return to parent

basecase:
# assign 0 as the value in $v0
# return to parent
sumton:
  beq $a0,$zero, basecase  # if n == 0 then branch to base case
  addi $sp,$sp,-8          # else, make space for 2 items on the stack
  sw $ra, 4($sp)           # store return address on stack
  sw $a0, 0($sp)           # store argument n on stack
                            # (will need it to calculate returned value)
  addi $a0, $a0, -1        # compute argument for next call: n = n-1
  jal sumton               # jump and link to sumton (recursive)
  lw $ra, 4($sp)           # load the return address
  lw $a0, 0($sp)           # load n from the stack
  addi $sp, $sp, 8         # change the stack pointer
  # register $v0 contains result of sumton(n-1)
  add $v0, $a0, $v0        # add n to $v0
  jr $ra                   # return to parent

basecase:
  addi $v0,$zero, 0        # assign 0 as the value in $v0
  jr $ra                   # return to parent
Frame Pointer ($30$)

Sometimes the size of stack frame is not fixed.

It may be useful to keep track of where frame starts.
main() {
    int m, n
    m = myfunction(n);
}

int myfunction(int j) {
    int k;
    return k;
}

* kernel calls main
Announcements

A2 is due next Sunday at midnight.

A3 will be posted soon and will be due on the last day of Reading Week (~3 weeks from now)

A4 (last one) will be posted in mid-March and due at end of March.

Last lecture is April 13. Final exam is April 26 (tent.)